Programming for the Java Platform Enterprise Edition

Enterprise Java Beans (EJB)

Opracował:
mgr inż. Tomasz Jastrząb

Table of contents

1 Introduction

Enterprise Java Beans (EJB) is a technology related directly to the Java Enterprise Edition platform, which unlike JPA requires the existence of a specialized server, dedicated specifically to this platform. Among others the servers providing Java EE support could be GlassFish, JBoss, WebSphere or WebLogic, but not Tomcat server. The description in the following sections and during the laboratory is related to EJB specification version 3.1.

1.1 A bit of history

The first version of EJB specification (EJB 1.0) was released around the year 1998, defining the EJB roles (e.g. bean provider, container provider, application assembler, etc.) and their responsibilities as well as the client view of EJB.

With the release of version 2.0 (in 2001) EJB technology started gaining popularity but also criticism, due to the level of complexity resulting from quite a large number of restrictions and requirements mostly for the bean provider. Among these restrictions one may name for instance the need for implementation of specific interfaces, the requirement of keeping appropriate method names, the necessity of the declaration of particular exceptions being thrown, and two level client view interfaces definition (home and component interfaces). Furthermore these interfaces could be local or remote.

The specification defined three types of components (beans):

- session beans, being responsible for implementing the business logic,
- entity beans, being the part of the persistence layer,
- message-driven beans, constituting the mean for asynchronous processing.

The persistence layer at this time was also different from the one defined in the EJB 3.0 specification, including Container Managed Persistence (CMP) and Bean Managed Persistence (BMP).

Moreover, all the dependencies and respective components' details were defined in external XML descriptor file (so called deployment descriptor), which being external to the code, and usually hard to verify and not that comfortable to create, could be also considered a drawback.

With the release of EJB 3.0/3.1 specification the persistence model has changed – Java Persistence API (JPA) was introduced and the entity beans were dropped (although they can be still used due to backward compatibility).

Moreover, thanks to the appearance of annotations, introduced in Java 5.0, the majority of configuration was moved to be performed using this mechanism and taking advantage of *Convention over Configuration* rule, i.e. intensive use of default values, without the need for their explicit specification.

Furthermore there are no longer separate home and component interfaces, which were replaced with business interfaces, which still may be local or remote. The specification also introduced the possibility of no-interface view creation, providing direct access to bean class.

Among the available bean types there remained session and message-driven beans, with the former being possibly stateless, stateful or singleton (details in section 1.2).

1.2 Components

As mentioned earlier, EJB 3.1 specification defines two types of components: session beans and message-driven beans. Apart from these two groups the specification mentions also entity beans, belonging to the previous EJB 2.0/2.1 version, being there the part of persistence model.

All component types "live" and operate on the server side, within the container, which provides among others the support for transactions, security or network communication. Usually, the role of the container includes also making decisions about creation and destruction time of various components, although in some cases the client can also partially influence this process (it is mainly related to stateful session beans).

Session beans Session beans are responsible for the implementation of business logic on the server side, accessible to clients using remote or local interfaces, or no-interface view. Depending on the available view the way components can be used differs, namely:

- **remote view** allows for the access from the same as well as different Java virtual machine (JVM), which means applications located on the server side and also those located elsewhere. The client can be another bean, web component, or an ordinary console application for the Java SE platform. In this case the client application uses only the interface without direct access to the bean class.

- **local view** allows for access only from within the same Java virtual machine, which enforces the collocation of the client and server side applications. In this case the client can only be another bean or a web

component located on the same server. Similarly to the remote view, only the interface is available, not the bean class itself.

- **no-interface view** means that the client can directly access the bean class, but it can only use the public methods. An attempt to access non-public methods results in *EJBException* exception. Due to the fact that no-interface view is a special form of local view, the client of this view can only be another bean or web component working on the same server.

The characteristic feature of local view is that parameters and return values are passed "by reference", whereas in case of remote view the data is passed "by value". It is a consequence of the need for network communication and data transfer, but it happens transaprently for the client.

Regardless of the view type, session beans can be created as one of the following three types:

- stateless session beans – they do not possess any state related to a particular client. It is because the container manages a pool of stateless session beans (usually smaller than current load, i.e. the number of clients) and successive method calls may be directed to different beans. Client application has no control over the component choice, neither it has the control over the process and time of bean creation or destruction.

- stateful session beans – they may contain some state related to a particular client, which should be preserved between successive business method calls. The client can control (partially) the time of bean creation and destruction through the methods marked with *@Init* and *@Remove* annotations.

- singleton session beans – according to the Singleton design pattern, these beans are created once and last through the whole application lifetime. The client may enforce that the creation time be during application startup, using *@Startup* annotation, but bean destruction happens automatically when the application is closed.

Message-driven beans Message-driven beans constitute the mechanism of asynchronous operation, but they do not provide any client view. They are based on messages' handling, placed in queues or topics to which the messages are assigned. Due to the lack of client view there is no possibility of returning any results to the client, message-driven beans are simply message

consumers. Depending on the type of messages which given bean is dedicated to, it should implement appropriate interface, whose method is called by the container when a new message arrives.

1.3 Annotations

The basic functionality related to session beans creation and use can be achieved thanks to only a few annotations from the javax.ejb package.

Depending on the client view the following annotations may be used *@Remote*, *@Local* and *@LocalBean*. They mean that the remote, local or no-interface view is created, respectively.

Depeding on the type of session bean one may use the following annotations *@Stateless*, *@Stateful* or *Singleton*, which respectively denote stateless, stateful and singleton bean. Each of these annotations allows for the specification of a name that may be further used in the process of bean searching. The name can be given using *name* or *mappedName* attributes, but the latter solution is considered as product-specific and not portable.

Nevertheless, a bean containing the mapping given by *mappedName* attribute can be found using JNDI[1], or made available through dependency injection. On the other hand, a mapping defined through *name* attribute can only be used with dependency injection, provided that the injection happens within the same application. The details of bean searching or dependency injection are given in section 1.4.

The process of dependency injection is performed by means of *@EJB* annotation, which in the default version recognizes proper bean based on its interface, provided that there exists only one implementation of this interface within the application and no mapping was used on the bean side. This annotation may be used for specifying the following attributes (among others):

- *mappedName* – it is a counterpart of the attribute with the same name given on the bean side in the annotations defining its type (stateless, stateful or singleton). Similarly to its counterpart it is treated as not portable and product-specific name.
- *lookup* – a portable alternative for *mappedName* attribute, allowing for finding a resource given in *mappedName* on the bean side.

[1]JNDI – Java Naming and Directory Interface, a structure of "folders" containing resources, which may be found basing on their names. An exemplary resource could be a data source or another bean.

- *beanName* – it corresponds to the *name* attribute in *@Stateless*, *@Stateful* and *@Singleton* annotations, allowing for bean recognition within the same application. It may appear instead of *lookup* attribute, but not together.

- *name* – specifies the name of given bean inside the environment of current component (details in section 1.4), may be later used with JNDI.

- *beanInterface* – specifies the class of bean interface, useful in case *@EJB* annotation is utilized together with *name* attribute to define bean mapping to be used later for JNDI lookup. It is important in case the type cannot be recognized based on the reference type.

1.4 Bean access

As mentioned earlier one may gain the access to enterprise beans in two ways – through dependency injection or JNDI lookup. The characteristic feature of both solutions is that for the beans with remote or local interface, injected/searched type is always an interface, not the bean class itself. Moreover, beans are never instantiated by the client (using the new operator), because their lifecycle is controlled by the container. Consequently, it is the role of the container to inject proper bean instances, or to make them available through lookup operation.

Dependency injection The process of dependency injection, may be performed through the declaration of appropriate class field, with the type matching bean interface's type and the use of *@EJB* annotation with proper attributes as given above. Alternatively, the declaration of bean injection can be placed on class level in which case *@EJB* annotation is used for giving the mapping name for the bean with specific interface, through *name* and *beanInterface* attributes. In such a case the bean may be further found inside JNDI structures in the client environment (i.e. using the path java:comp/env/). It may happen in the case the client is another bean or a web component located on the same server.

JNDI lookup To be able to look up JNDI structures, a context needs to be obtained, constituting the starting point of the lookup operation. To do so, an object of *javax.naming.InitialContext* type needs to be created. Next, using the context, *lookup* method should be called, which as an argument takes the name under which the resource was mapped (with the use of *mappedName*

attribute in annotations specifying bean type, or *name* attribute of *@EJB* annotation).

In the case the mapping is a global name (defined with *mappedName* attribute) it can be given directly to *lookup* method, otherwise a local resource is searched, so the name should be prefixed with the following path java:comp/env/. Regardless of specified name, the result of lookup operation has to be cast to the interface type of searched bean (since *lookup* method returns a result of type *Object*) or directly to bean type in case of no-interface view. Both the creation of *InitialContext* and the use of *lookup* method enforces the handling of *javax.naming.NamingException* exception.

2 Laboratory

The description enclosed in this section is based on Netbeans 7.1, although the tests performed on newer version 7.3 were also successful.

2.1 Project configuration

To have the ability of enterprise application project creation and deployment it is necessary to possess a server supporting Java EE platform. Potential servers were already mentioned in section 1. Further description assumes the use of GlassFish 3.1.1 server.

Four projects are required for the enterprise application to work properly, with three of them constituting the interrelated parts working on Java EE platform, and the fourth one being an ordinary Java SE application used for holding and compiling remote resources. To create an enterprise application one shall choose menu ***File*** → ***New Project...***, and then *Java EE* → *Enterprise Application*.

This type of project allows for creation of the superior unit, containing relationships with the other parts of business application as well as EJB Module containing beans and Web Application Module. During the laboratory, out of the last two mentioned items, only the EJB module will be created, since the client will be a console application. To do so, after specifying project name, choosing GlassFish server and Java EE 6 (or higher) platform, one should check the option *Create EJB Module* and click **Finish**. As a result of these actions, two new projects should be created.

The next step is the creation of client application, by choosing once again ***File*** → ***New Project...***, and then *Java EE* → *Enterprise Application Client*. Once again it is necessary to specify project name, select compatible GlassFish and Java EE platform versions and later to choose from the *Add to Enterprise Application* combobox the enterprise application created in the previous step.

The final step of project configuration is the creation of an ordinary console application, through menu ***File*** → ***New Project...*** and *Java* → *Java Application*. The option creating main class for this project (*Create Main Class*) may be unchecked.

2.2 Session beans generation

To create a session bean one should use right mouse button (rmb) on the level of EJB Module's *Source Packages* directory (the module's name will by default end with *-ejb*) and select ***New*** → ***Session Bean***. In the dialog

that appears one shall specify the name of created bean (inside *EJB Name*) and the package where the bean should be located (*Package*). Furthermore in the section captioned *Session Type* it is necessary to specify the bean type and choose a client view in the *Create Interface* part. In case no element is checked in the latter item no-interface view will be created. Creating a bean with remote view, it is required to specify the project (ordinary one, not related to current business application), which will contain generated interface. Confirming the choices with **Finish** button, depending on the interface type selected, either an interface and a bean or only a bean will be created.

Having the skeleton for the session bean, one may add business logic to it, which in the form of a method header will be also automatically added to corresponding interface. To do so, inside the bean class one should click **Alt** + **Insert** and select ***Add Business Method*** option. In the dialog that appears, method name, return type and possible parameters may be specified (fields *Name*, *Return Type* and **Add** button in *Parameters* tab, respectively). The choices should be confirmed using **Finish** button.

After implementing business methods, created session bean may be called from the client application (or inside other bean in case of local client view). This may be done by clicking **Alt** + **Insert** inside the *main* method of the main client class (by default called *Main*), and choosing ***Call Enterprise Bean***. Next, appropriate bean should be located and its choice should be confirmed with **OK** button. As a result, the dependency on the specified bean will be injected in the form of a class field. Alternatively, the bean may be found manually using JNDI as described in section 1.4.

2.3 Execution

To run the project one shall perform the following steps:

1. Click rmb on the console project level and choose ***Clean and Build***. Alternatively, being at the console project level, **Clean and Build** button may be clicked (the one with hammer and broom icon).

2. On the level of enterprise application client and EJB Module, unfold the contents of *Libraries* folder, right-click on the JAR file with the name of console application and choose ***Remove...***. After removal right-click the *Libraries* folder once again, select ***Add JAR/Folder...*** option and show the location of the previously deleted JAR file inside console application.

3. On the level of main enterprise application (the superior unit) use rmb and choose ***Run***. As a result the GlassFish server should start, and the

application shall be deployed on server.

4. Finally, marking the enterprise application client project, using rmb and choosing ***Run*** option again, the application should start successfully after a while.

In case of some errors related to being unable to delete a JAR file coming from the console application, it is necessary to start the Task Manager and kill both processes named java.exe. Next, the aforementioned file should be deleted manually from disk, and the steps given above shall be repeated. It may be also necessary to manually start the GlassFish server through ***Services*** → ***Servers*** → ***rmb*** → ***Start***.

In case of errors related to the *NamingException* exception the correctness of provided mappings should be verified, together with proper project depoyment on server (saving any change in the bean classes, the project should be automatically refreshed on server ((observe a message saying <project name> deployed in the bottom left part of the screen)). After the project is successfully deployed step no 4 should be repeated.

3 Tasks

The tasks mentioned in this section are supposed to be performed during the laboratory classes, except for task no. 4 which defines the requirements for the report.

3.1 Task no. 1

Compare the stateless, stateful and singleton session beans. Create a session bean with a remote interface having a single integer field, constituting bean's state. The field should store the total of operations performed in the only business method of the bean, performing the multiplication of integer numbers. In the client application use JNDI lookup mechanism to obtain a reference to the bean.

The scenario of tests should be as follows:

- create *InitialContext* object and look up the session bean,
- perform the multiplication twice and display the total of these operations,
- look up the bean again and assign to another reference,
- perform the multiplication twice and display the total of these operations,
- create *InitialContext* object once again and look up the session bean,
- perform the multiplication twice and display the total of these operations.

After each run of the experiment change the declaration of session bean type and run the test once again. Compare the results and draw some conclusions.

3.2 Task no. 2

Create a session bean with local client view providing the methods performing addition, subtraction and mutliplication of integer numbers are returning the results as integers.

Create a session bean with remote client view, being a matrix calcualtor, allowing for addition, subtraction and multiplication of matrices as well as multiplication by scalar. To perform respective steps of these operations use

previously created session bean with local interface. The bean should be injected in the way allowing for specifying its name and later found in the local environment (see section 1.4).

The remote bean should be used in client application through dependency injection.

3.3 Task no. 3

Create a class *Customer* of JavaBean type (not to confuse with Enterprise Java Bean) possessing the fields *id*, *name*, *companyName* and *yearlyIncome*. The class should implement *java.io.Serializable* interface, and contain a constructor with parameters, getters and setters for all the fields as well as overriden version of *toString()* method displaying all fields. The class should be placed in the console application part of the project (i.e. Java Application).

Compare the way the local and remote interface beans work. To do so, create session beans with remote and local client view, implementing the following method *Customer findRichest(List<Customer> customers)*, which finds the richest customer based on *yearlyIncome* field.

The implementation in the local bean should find the richest customer and modify the *name* field replacing its contents with the text "I'm the richest". The remote bean should display the contents of customer list, call the method of local session bean, display the list once again and return the result found.

Client application shall use the remote client view, and similarly to the remote bean should display list contents before and after method call.

Warning! The results of data display on the server side will be shown in GlassFish server's console window, not the standard console window – tab named *GlassFish 3.1.1*, not *<application name> (run)* in the *Output* window!

Compare obtained results and draw some conclusions.

3.4 Task no. 4

Create an enterprise application using session beans for performing CRUD operations on the database using JPA mechanisms. Use the database schema created in the previous report.

- application client should be a console one,
- respective CRUD operations should be done in separate methods,

- data constituting the contents of entity classes should be provided by the user,
- erroneous data should be detected and corrected before performing database operations,
- session beans may be accessed through dependency injection or JNDI lookup,
- report should contain Javadoc documentation for all created elements (classes, fields, methods),

The report should consist of three folders: *bin*, *doc*, *src*. *bin* folder should contain an *.ear file containing the enterprise application. *doc* folder should contain generated Javadoc documentation including *@author* and *@version* tags on the class level as well as private and package private members. *src* folder should contain source codes with packages. Inside *doc* and *src* folders the elements related to every part of the application should be placed (in separate subdirectories), including bean-related part, application client and the console part as well.

Identifiers of created classes, fields and methods should be written in English and comply with Java naming convention.